A Brief Introduction to Design Patterns

Jerod Weinman
Department of Computer Science
Grinnell College
weinman@grinnell.edu

Contents

Chapter 1

Introduction

As you have probably experienced by now, writing correct computer programs can sometimes be quite a challenge. Writing the programs is fairly easy, it can be the "*correct*" part that maybe is a bit hard. One of the prime culprits is the challenge in understanding a great number of complex interactions. We all have probably experienced the crash of some application program or another, and there are examples of epic (and plenty of non-epic!) software projects that never even got off the ground, such as the FBI's Virtual Case file. How is it that other fields, such as structural engineering that gives us vast bridges and tall buildings, have mastered problems of such great complexity? Bridge and building failures are drastically less common than software failures.

Arguably, one reason is those other fields have identified the commonalities amongst their solutions to various problems and created patterns that inform their design of solutions to new problems. These *patterns* are useful abstractions. We recognize patterns and similarities frequently in our every day lives. For instance, a pickup truck is like a car (they are both automotive modes of transport), though a truck often has fewer seats and a cargo bed. There are also analogies in role; a little fish is to a shark as a zebra is to a cheetah (i.e., food or prey). People are very skilled at finding abstract concepts and patterns in our every day lives, and we can put the the notion of design patterns to work for us in object-oriented programming (OOP).

The OOP paradigm has been a useful development in solving problems with computers because it enables us to closely model the task at hand. Dealing with objects means we are thinking about the agents in our computational world, the data they manage, and the collaboration or interactions among them. The utility of design patterns is not about the creation of isolated objects, but about the communications and roles among the objects. The point is to catalogue common roles and methods of communication.

While there are many design patterns being proposed and cataloged all the time, the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides gives a common set of patterns [1]. *Design Patterns* catalogs 23 major patterns and puts them into three basic categories: creational, structural, and behavioral.

Creational patterns create objects for you. These give you some flexibility in using different types of objects in different cases. For instance, you don't have to assemble your own value meal when you visit a fast food restaurant. There are different types that all satisfy the same goal of giving you a meal (with modest nutritional value). Rather, an employee takes your request and assembles the meal for you. This is an example of a *builder* pattern.

Structural patterns are about composing several objects into larger structures. One simple example is like a power adapter. European wall plug power outlets have two round prongs and give electricity that is 220V. An unfortunate North American may have an appliance whose plug will not fit and will expect a measly 110V. Thus, an adapter is needed so that every traveler need not worry about how to bridge this electrical divide herself. This is an example of one of the simplest structural patterns, called (not surprisingly) an *adapter*.

Finally, **behavioral** patterns are about communication and flow control. One common example is someone who subscribes to a magazine. The subscriber can't (or shouldn't) follow all of the details of what subject they might be interested in, but periodically they receive a succinct update of the relevant information. This type of pattern is called an *observer*, in which automatic status updates about one object are sent automatically to another object (the subscriber).

In conclusion, it is important to note that patterns are not created. Rather, they are *discovered.* The design patterns catalogued by Gamma, Helm, Johnson, and Vlissides and many others since were culled from a great deal of experience building object-oriented software. For this reason, studying design patterns can seem somewhat strange at first. Without sufficient OOP experience, the utility of the abstractions that patterns represent may not be immediately apparent. However, our hope is that early exposure, reflection, and additional experience will convince you of their power. If nothing else, once you begin designing more complex object-oriented programs, we hope some day you recognize the patterns inherent in your problems.

Chapter 2

Creational Design Patterns

2.1 Introduction

As their name suggests, the family of creational patterns all deal with creating instances of objects. The OOP principle of polymorphism allows our programs to work flexibly with many different specific types of objects that share particular properties. The specific, runtime object type is sometimes unimportant and depends on the situation. We can extend this idea to the act of creating objects as well.

We already know how to create objects in Java. For instance,

```
w = new Widget();
```

uses the `new` operator. However, this fixes (or hard-codes) the method of creating the object to which the variable `w` refers. By now you know we try to avoid hard-coding values in our programs. We can extend this idea in the OOP framework by also making the mechanism for creating objects vary depending on the context. In this sense, abstracting object creation into special classes that control object creation enhances the flexibility of our programs.

While there are several standard creational patterns, we will consider four: factory, abstract factory, builder, and singleton. Let us look at each of these in more detail.

2.2 Factory

The simplest way to think of a factory is a generalization of the object constructor. Factory patterns allow two important types of variation in object creation. First, we may want to generalize the mechanism for creating an object of a particular type. In addition, the runtime type of the object to create might vary, depending on some particulars.

In the first case, Java's overloaded class constructors do not always clearly specify the means of creating an object. As an alternative, a factory method can be used to return an object of a particular type, rather than calling the constructor directly.

For instance, complex numbers can be represented in one of two ways. First, we might specify their Cartesian real and imaginary components, as in $3 + 4i$. Here, the number 3 is the real component, while the number 4 is the imaginary component. Alternatively, we could specify the same

quantity in polar coordinates, with a radius (or magnitude) and an angle of direction. In this case, the radius would be 5, while the angle is $\tan^{-1}(4/3) \approx 53.1°$. In any case, we could construct a complex number in one of two ways, but the key problem is that *both* ways require two doubles as input to specify the number. If a constructor takes two doubles, are these to be the Cartesian or polar coordinates?

We could disambiguate their meaning by using a constructor that takes an extra boolean flag to indicate one or the other, but this seems inelegant and prone to misunderstanding. Instead, we make the constructor private and use a clearly named factory method for each representation, as shown in Figure 2.1. [4]

This first type of factory method, given by `fromCartesian` and `fromPolar`, varies the mechanism by which an object is created.

The second generalization a factory method might make takes advantage of polymorphism. For example, Java's `Number` class is abstract, therefore no program can directly instantiate it. However, `Number` does have several methods that all of its subclasses support. Two standard subclasses are `Integer` and `Double`, and we just added a `Complex` descendant of `Number` to the mix in the previous example. Say we wanted to read a number from some input stream. We may want to handle any type of number—integer, double, or complex—however, from our program's standpoint, we may only care that the type is a number. Because the code for disambiguating the input is somewhat involved and may be generally needed by many clients, we can farm that task out to a factory for the `Number` class that instantiates the appropriate type based on input from a `Scanner`, as in the example of Figure 2.2.

This is the most common type of factory: it instantiates an arbitrary member of a class hierarchy based on the situation at hand. While the runtime type varies, the static type is a common member of the class hierarchy.

Cooper [2, p. 31] gives the following advice on when to use a factory method:

- A class can't anticipate which kind of class of objects that it must create.
- A class uses its subclasses to specify which objects it creates.
- You want to localize the knowledge of which class gets created.

2.3 Abstract Factory

An abstract factory takes the same idea one level higher. Instead of having a simple factory that returns a polymorphic reference from a class hierarchy, an abstract factory provides a hierarchy of factories with a common theme. For example, we could define a factory class having methods to create instances representing user interface widgets.

```
public abstract class AbstractWidgetFactory
{
  public Button getButton();
  public TitleBar getTitleBar();
  public ScrollBar getScrollBar();
}
```

```
public class Complex extends Number
{
  /** Create a Complex number from Cartesian coordinates
   *
   * @param real The real component of the number
   * @param imag The imaginary component of the number
   */
  public static Complex fromCartesian(double real, double imag)
  {
    return new Complex(real, imag);
  }

  /** Create a complex number from polar coordinates
   *
   * @param radius The radius of the polar representation
   * @param angle The angle of the polar representation
   */
  public static Complex fromPolar(double radius, double angle)
  {
    return new Complex( magnitude * Math.cos(angle),
                        magnitude * Math.sin(angle) );
  }

  /** Create a Complex number from Cartesian coordinates.
   *  Note the constructor is private to avoid ambiguity.
   *
   * @param real The real component of the number
   * @param imag The imaginary component of the number
   */
   private Complex(double real, double, imag)
   {
      // ...
   }

// ...

}
```

Figure 2.1: A class with a private constructor and public factory methods to disambiguate parameter meaning.

```
public class NumberFactory
{
  /** Decides which type of Number to return based on the
   *  current line of the given scanner.
   *
   * @param s Scanner to read input from
   * @returns Number read from scanner
   * @throws NumberFormatException if no number can be parsed
   */
  public static Number fromScanner(Scanner s)
    throws NumberFormatException
  {
    /* Do we have a complex number? a + b * i */
    if (s.hasNext("\\d+(\\\\.*\\d)? +\\+" +  // Real part +
                  "\\d+(\\\\.*\\d)? \\* i")) // Complex part
    {
      double real = s.nextDouble(); // Read real part
      s.next();                     // Skip + (plus)
      double imag = s.nextDouble(); // Read complex part
      s.next();                     // Skip * (times)
      s.next();                     // Skip i

      return Complex.fromCartesian(real,imag);
    }
    else if (s.hasDouble())
    {
      return new Double(s.nextDouble());
    }
    else if (s.hasInt())
    {
      return new Integer(s.nextInt());
    }
    else
      throw new NumberFormatException();
  }
}
```

Figure 2.2: A factory class that uses polymorphism to vary the runtime type of object returned depending on the input.

The `Button`, `TitleBar`, and `ScrollBar` classes would also necessarily be abstract. With an interface like this, a program client may write a graphical program that requires only a polymorphic `AbstractWidgetFactory` reference to create its GUI elements. Such a program could then run portably across a variety of platforms, so long as they have concrete versions of the `AbstractWidgetFactory`.

Suppose we do have two concrete factories, `MacWidgetFactory`, and `X11WidgetFactory` that both extend `AbstractWidgetFactory`. Assume that `MacButton` and `X11Button` are both concrete subclasses of `Button`. The `MacWidgetFactory` method `getButton` would return a (runtime type) `MacButton`. The abstract factory interface is therefore satisfied because a `MacButton` *IS-A* `Button`.

The main point of this pattern is that the client is indifferent to the details of the concrete classes. These differences are transparent to a program having a reference to an `AbstractWidgetFactory` object.

2.4 Singleton

While the singleton pattern falls in with other creational patterns, it is somewhat distinctive in that it limits creation of objects. A singleton restricts a class to one and only one instance, providing a single, global means of accessing that instance. Cooper [2, p. 39] gives a few examples: "your system might have only one window manager or print spooler or one point of access to a database engine."

However, singletons have their drawbacks and critics. In particular, they introduce a global state, which can complicate unit testing. Singletons may also pose problems for parallel, multithreaded applications that would require serialized locking of member data, which limits parallelism. The programmer is advised to be sure that a singleton model is truly appropriate for their problem domain.

Creating a singleton is rather straightforward in Java. One would make the constructor private and store the singleton object in a private static variable. One must then add a public static function that checks the variable, constructs an instance if none exists, and otherwise returns a reference to the singleton object.

2.5 Builder

The final creational pattern we'll study is the so-called builder pattern, which allows us to abstract the algorithm for creating an object in a slightly different way from the factory pattern. In the builder pattern, the essence of the construction algorithm remains the same; that is we always use the same general algorithm to build an object. However, the components may differ. In essence, a builder constructs objects from components.

A builder pattern has several participant classes: [3]

Builder provides an interface for creating objects (the products)

Concrete Builder provides an implementation that constructs the individual components of the product

Director uses a Builder to retrieve components and assemble them appropriately

Product is the object created by the Director using parts from a Builder

Cooper [2, pp. 55–56] points out the following consequences of a builder pattern

1. A Builder pattern lets you vary the internal representation of the product that it builds. It also hides the details of how the product is assembled.

2. Each specific Builder is independent of any others and the rest of the program. This improves modularity and makes the addition of other Builders relatively simple.

3. Because each [Director] constructs the final product step by step, depending on the data, you have more control over each final product that a [Director] constructs [using a Builder].

He goes on to mention this "is somewhat like an Abstract Factory pattern in that both return classes made up of a number of methods and objects. The main difference is that while the Abstract Factory returns a family of related classes, the [Director] constructs a complex object step by step depending on the data presented to it" (p. 56).

Chapter 3

Structural Design Patterns

3.1 Introduction

Structural design patterns are about composing classes, interfaces, and objects into larger structures. There are two categories of structural patterns, *class* patterns and *object* patterns. A class structural pattern uses inheritance to create new, more useful classes. These use the *IS-A* relation. In contrast, an object structural pattern composes objects to create entirely new type with the *HAS-A* relation. Some of the simple structural patterns we will look at include the adapter, façade, proxy, flyweight, and decorator patterns.

3.2 Adapter Pattern

As described in the introduction, the adapter pattern allows one to convert the interface from one object into another interface that may be more useful. This notion is fairly straightforward. We create a new class that has the interface needed and use it to communicate with—or translate to—the interface of an existing object.

There are two basic ways to accomplish this interface adaptation: inheritance or object composition. The first way involves extending the existing class and then adding the methods that conform to the new interface. This is called a class adapter, and you should notice that it is a class pattern. The second way would have an object of the existing class as a field in the new class. This new class can then pass along method calls to the object with any appropriate translations.

Let's look at a classic example. Say we have two types of holes, round and square, and the interface for inserting each of these objects is different:

```
public class SquarePeg
{
  public void insert(String msg)
  {
    System.out.println("Square peg: insert" + msg );
  }
}
```

```
public class RoundPeg
{
  public void insertInto(String msg)
  {
    System.out.println(“Round peg: insert” + msg );
  }
}
```

We know square pegs will not fit into round holes, so we can build an adapter for the square peg (using composition) that conforms to the (different) interface of the round peg.

```
public class SquareToRoundAdapter extends SquarePeg
{
  private RoundPeg rpeg;

  public SquareToRoundAdapter(RoundPeg peg)
  {
    this.rpeg = peg;
  }

  public void insert(String msg)
  {
    rpeg.insertInto(msg);
  }

}
```

We can demonstrate this simple adapter with the following code:

```
RoundPeg rpeg = new RoundPeg();
SquarePeg speg = new SquarePeg();

speg.insert(“square”);

SquareToRoundAdapter apeg = new SquareToRoundAdapter(rpeg);

apeg.insert(“round?”);
```

Thus, `SquareToRoundAdapter` *IS-A* `SquarePeg`, but it *HAS-A* `RoundPeg`, allowing us to convert from the square peg interface into a round peg interface.

3.3 Façade

According to the Oxford American Dictionary, a façade is “the face of a building.” Figuratively, however, it is “an outward appearance that is maintained to conceal a less pleasant or creditable

reality." This definition applies reasonably to its use as a design pattern. In a nutshell, the façade encapsulates a set of complex classes into a simpler interface. One does not have to look far to see the utility of this design pattern.

Whereas detailed control is sometimes necessary in a large complex system, it may not always be desired by the average programmer. Consider I/O programming in the `C` programing language on a *nix system. On the one hand, you could write a device driver to do it, but this is painful and unnecessary. After all, you really do not want to worry about spinning up disk platters and moving read heads when all you want to do is write the next number to a file. Thus, the operating system provides you with a beautiful abstraction called `write`. No more worrying about disk sectors; instead, all you need to do is provide a file pointer, a character buffer and the number of bytes to write. However, for many people this is still too much control. Who wants to keep track of bytes? Thus the standard C library provides an even more beautiful abstraction called `fprintf` that only requires a file stream, a straightforward control string, and your input.

The chasm between disk platters and the control string "`%d`" for doubles is appreciable. The `write` procedure is part of the system call interface, which is a façade for some very unpleasant levels of control. Furthermore, the `fprintf` procedure is part of the standard `C` library, which provides a façade for the system calls. Though neither of these are object-oriented, the notion of abstraction is the same.

Cooper notes two important consequences of the façade pattern:

> [It] shields clients from complex subsystem components and provides a simpler programming interface for the general user. However, it does not prevent the advanced user from going to the deeper, more complex classes when necessary.
>
> In addition, the Façade allows you to make changes in the underlying subsystems without requiring changes in the client code ... [2, pp. 120–121]

3.4 Flyweight

Good object-oriented design often uses many small, similar classes. A program might have lots of instantiated objects from these classes. When the similarities can be explicitly shared among these objects, we have the opportunity to use the flyweight pattern.

Named for the lightest boxing weight class (contrasting starkly with heavyweight), the flyweight pattern separates an objects' fields into *intrinsic* and *extrinsic* data. The intrinsic data is unique to the instance, while the extrinsic data is shared among many objects. The overhead of the system is thereby reduced when many objects point to a very small number of shared objects. In Java, most objects have extrinsic data because all reference variables are essentially pointers, and multiple variables can refer to the same reference object. You may have established a flyweight pattern without even knowing it. However, the flyweight is not as trivial or immediate in other object-oriented programming languages, such as `C++`, which distinguish objects and object pointers.

The Java language itself builds in the flyweight pattern in another, less explicit way. Consider two `String` variables both initialized to the same string constant.

```
String one = "one";
String won = "one";
```

To do something similar in C, you would declare these as `char*` variables and separate space would be allocated on the stack for each string. Thus, it seems obviously silly in C to test string equality with the expression

```
one == won
```

but some novice Java programmers still write this. Does the Java compiler handle this any differently from C? The answer is yes and no. Because `String`s are reference variables, we have no guarantee that two references considered `equal` (as `one` and `won` would be) should actually refer to the same object (as `one==won` asserts). In this regard, the Java compiler gives no general special treatment for `String` references.

You have probably already noted that `String` is an important and frequently used type. In fact, if you have done any text processing in Java, you have likely created a set of classes that have `String` variables all over the place, and many of these could contain the same sequence of characters. If we are processing very large chunks of data (as your friendly Internet search engine does), we certainly don't want to store all those identical strings in separate memory locations. Instead, we'd prefer to store the string just once and have references to that memory location in several places.

Fortunately, Java *does* in fact allow a way to make sure we use the same memory location. The `String` member method `intern()` returns a "canonical representation" of a string object, which is a flyweight pattern. According to the official Java API specification:

> A pool of strings, initially empty, is maintained privately by the class `String`.
>
> When the intern method is invoked, if the pool already contains a string equal to this `String` object as determined by the `equals(Object)` method, then the string from the pool is returned. Otherwise, this `String` object is added to the pool and a reference to this String object is returned.
>
> It follows that for any two strings `s` and `t`, `s.intern() == t.intern()` is `true` if and only if `s.equals(t)` is true.
>
> All literal strings and string-valued constant expressions are interned. String literals are defined in §3.10.5 of the Java Language Specification [7]

There you have it. Rather than worrying about whether an object A has a reference to variable `one` from above and object B has a reference to variable `won`, we can use a flyweight by storing `one.intern()` and `won.intern()` to be sure that both refer to the same object.

3.5 Proxy

Though there are others, our final structural pattern is the proxy. This pattern represents a complex or resource consuming object with something simpler. In essence, it is a stand-in object that has the same interface as the object it stands in for.

One example is creating an object that simply represents a very large image in a web page. You typically do not want to wait for an image to load before reading the page you are visiting or seeing the general layout. Thus, until the actual image is loaded, the web browser's object hierarchy may

return a proxy for the image that contains only the height and width and whose display method simply gives an empty rectangle. The proxy can then work in the background to load the image and begin displaying it properly only when the download is complete.

Another common use of a proxy is to delay expensive operations until absolutely necessary. One example is creating copies of very large objects. If the copy is never altered, it makes sense to only share a reference, in order to conserve memory. However, if the copy is changed, we may have to make a complete and bona fide replication of the data so that the original remains unaltered. This behavior is often called copy-on-write.

3.6 Decorator

The decorator pattern modifies the behavior of objects without a derived class. In other words, a decorator class acts as a wrapper around some existing class. This is particularly important when a large number of independent extensions could cause a multiplicative explosion of subclasses to support all possible combinations. For instance, consider a simulation program with five actor objects. If three of these objects needed an additional feature, you might derive a new subclass to add that feature. However, if the features were all different, you would need three separate subclasses. Imagine that, in fact, a fourth object needed features from two of these other subclasses. There is no clear, clean way to solve this combination problem with inheritance. Sanders and Cumaranatunge point out:

> Sometimes when you think about key OOP concepts such as *inheritance*, you have to consider its larger consequences. The upside of inheritance is that once a superclass has been established, all the subclasses inherit its features. However, you may not want or need all the "stuff" for every subclass from a superclass. If you want to add functionality to an object, subclassing may not be the way to go, because everything else subclassed from the same superclass may be unnecessarily burdened by unused functionality. [5, p. 132]

Enter the decorator. It is also known as the *wrapper* pattern. The notion of "wrapping" is central to this design pattern. "Unlike subclassing, which extends one class into another, wrapping allows one object to use another's characteristics without extending either the wrapped object or the object doing the wrapping" [5, p. 133].

How does this work? The general class hierarchy is illustrated in Figure 3.1 . At the root, we need an abstract component class from which every other class is derived. This component describes the basic behavior by specifying the required operation(s). Concrete component classes are then derived from the abstract component. These are the objects that will get decorated. Think of the concrete component as the innermost layer of a Russian *matryoshka* nesting doll [5, pp. 131-132].

We then create an abstract decorator class that also extends the component. The most important capability of the abstract decorator is that it can now override methods of the original component. In addition, this abstract decorator *HAS-A* abstract component. The abstract decorator overrides the component operations, which can then also pass along messages as appropriate to its member field, the abstract component.

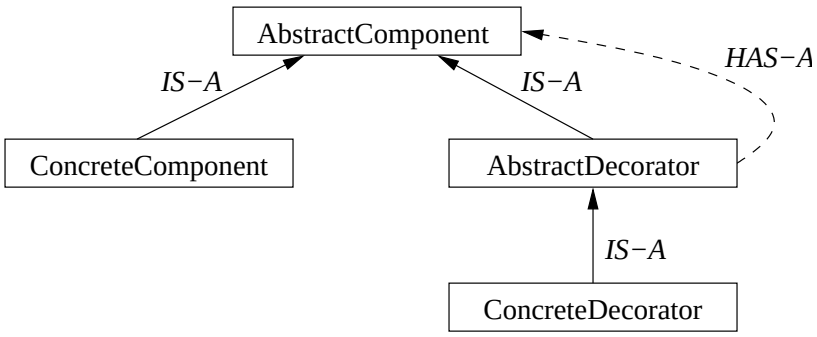

Figure 3.1: Class hierarchy for the decorator pattern.

Finally we derive concrete decorators. The concrete decorator adds functionality to a component. That is, it is the object that actually does the decorating. How? Note that the concrete decorator also *IS-A* abstract component. Therefore it intercepts messages for the component interface, adding its computation before passing the message along to the member component via its parent, an abstract decorator. Note that the member component could itself be another decorator (because a decorator *IS-A* component). In this way, we can wrap arbitrary functionality on top of a component. Each decorator object is like another Russian doll wrapped around the previous layer.

Figure 3.2 illustrates an example abstract component that represents a pizza, which has two simple operations. Our concrete components reflect the different types of pizza, such as deep dish style. We also declare an abstract decorator that can add toppings to any type of pizza, no matter the style or other toppings. Figures 3.3-3.5 illustrates a few simple concrete decorators that actually add the toppings. An example program and output are given in Figure 3.6.

```
public interface Pizza
{
  double getCost();
  String getIngredients();
}

public class DeepDishPizza implements Pizza
{
  private static final double cost = 4.95;

  public double getCost() { return cost; }

  public String getIngredients()
  {
    return "Deep-Dish Pizza pie";
  }
}

public abstract class PizzaDecorator implements Pizza
{
  private Pizza base;

  public PizzaDecorator(Pizza base)
  {
    this.base = base;
  }

  public double getCost() { return base.getCost(); }

  public String getIngredients()
  {
    return base.getIngredients();
  }
}
```

Figure 3.2: Essential elements for a decorator pattern: an abstract component `Pizza`, a concrete component `DeepDishPizza`, and an abstract decorator `PizzaDecorator`.

```
public class CheeseDecorator extends PizzaDecorator
{
  private static final double cost = 1.25;

  public CheeseDecorator(Pizza pizza)
  {
    super(pizza);
  }

  public double getCost()
  {
    return super.getCost() + this.cost;
  }

  public String getIngredients()
  {
    return super.getIngredients() + ", Cheese";
  }
}
```

Figure 3.3: Concrete cheese decorator for an abstract pizza component.

```
public class PepperoniDecorator extends PizzaDecorator
{
  private static final double cost = 2.00;

  public PepperoniDecorator(Pizza pizza)
  {
    super(pizza);
  }

  public double getCost()
  {
    return super.getCost() + this.cost;
  }

  public String getIngredients()
  {
    return super.getIngredients() + ", Pepperoni";
  }
}
```

Figure 3.4: Concrete pepperoni decorator for an abstract pizza component.

```
public class MushroomDecorator extends PizzaDecorator
{
  private static final double cost = 1.75;

  public MushroomDecorator(Pizza pizza)
  {
    super(pizza);
  }

  public double getCost()
  {
    return super.getCost() + this.cost;
  }

  public String getIngredients()
  {
    return super.getIngredients() + ", Mushrooms";
  }
}
```

Figure 3.5: Concrete mushroom decorator for an abstract pizza component.

Some actual examples of the decorator pattern include layers of GUI widgets (quite literally decorators of of graphical components) and the `java.io` classes (c.f. Weiss [6, §4.5.3]).

Finally, Cooper makes the following comparisons among some structural design patterns:

> Both the Adapter and the Proxy patterns constitute a thin layer around an object. However, the Adapter provides a different interface for an object, where as the Proxy provides the same interface for the object but interposes itself where it can postpone processing or data transmission effort.
>
> A Decorator also has the same interface as the object that it surrounds, but its purpose is to add additional (sometimes visual) function to the original object. A Proxy, by contrast, controls access to the contained class. [2, p. 137]

```
public class Driver
{
  public static void main(String[] args)
  {
    Pizza pizza = new DeepDishPizza();

    System.out.println(pizza.getIngredients() +
       " $" + pizza.getCost());

    pizza = new CheeseDecorator(pizza);

    System.out.println(pizza.getIngredients() +
       " $" + pizza.getCost());

    pizza = new MushroomDecorator(pizza);

    System.out.println(pizza.getIngredients() +
       " $" + pizza.getCost());

    pizza = new PepperoniDecorator(pizza);

    System.out.println(pizza.getIngredients() +
       " $" + pizza.getCost());

  }
}
```

```
Deep-Dish Pizza pie $4.95
Deep-Dish Pizza pie, Cheese $6.2
Deep-Dish Pizza pie, Cheese, Mushrooms $7.95
Deep-Dish Pizza pie, Cheese, Mushrooms, Pepperoni $9.95
```

Figure 3.6: Example use of decorators (top) and sample output (bottom).

Chapter 4

Behavioral Design Patterns

4.1 Introduction

Whereas creational design patterns have to do with instantiating objects and structural patterns have to do with composing objects and classes into larger, more complex structures, behavioral design patterns are concerned with communication among a family of objects. Because computation happens primarily via communication in object-oriented programming, behavioral patterns are a fundamental family of designs. While there are a variety of behavioral design patterns, we will consider four here: chain of responsibility, observer, state, and visitor.

4.2 Chain of Responsibility

A chain of responsibility establishes a pattern for handling commands or requests. We connect a sequence of processing objects that each determine whether it is responsible for processing some command. If the current processor cannot, it passes the command to the next link in the chain.

In essence, the chain of responsibility is a linked list where each node is endowed with additional logic for filtering commands as well as the means for handling commands it intercepts. A common example would be the event handler for a GUI. For instance, clicking the mouse can happen anywhere on the screen, but which program or window should receive the details of that click event? Typically this is determined via a chain of responsibility, where each link in the chain would determine whether it was the ultimate recipient responsible for processing the event. If not, the event would be passed to the next link. The request handler is often decoupled from the chain so that client programmers like you are responsible only for determining how to deal with an event, while the GUI manages the chain and links on your behalf. This decoupling is a hallmark of good OOP practice.

4.3 Observer

The observer pattern handles another type of communication: delivering status updates to objects. Consider a spreadsheet with several graphs of data in cells. You probably expect that if you change a value in one of the cells, the graph would reflect your new data. How can a program guarantee

this? We need each component of the graph to connect somehow to the data it is derived from. However, no one would buy your spreadsheet program if creating a graph caused an endless waste of CPU cycles because your graph component was constantly reading the cell data to make sure it is updated. Instead, we reverse the responsibility. Any time a graph component is derived from a cell, it sets up a subscription to the cell wanting to be notified of any updates.

The observer pattern establishes a one-to-many dependency between a "subject" and its dependent followers. Once the connection is established, the subject is responsible for informing dependents of any state changes. Dependents can then react to these changes in whatever manner is appropriate.

For example, consider how Twitter pushes updates to clients that are subscribed to particular feeds. To implement the observer we need an abstract `Subject` class, which represents the source Twitter feed or spreadsheet data cell.

```
public abstract class Subject
{
  ArrayList<Observer> dependents = new ArrayList<Observer>();

  public void subscribe(Observer o)
  {
    dependents.add(o);
  }

  public void notifyObservers(Message m)
  {
    for (Observer o : dependents)
      o.notify(m);
  }
}
```

`Subject` objects keep a list of subscribing dependents and allows sending a message notification to all of them. In an observer, we might have the following lines to establish a subscription:

```
Subject feed = ...
feed.subscribe(this);
```

Thus, subjects and observers are only loosely coupled. As Cooper [2, p. 216] points out, when many small changes are made to the data, we have a concomitant flood of notifications from subject to observers. This can be a disadvantage if the cost of updating the observers is high. Therefore, we may want to introduce "some sort of change management so that the Observers are not notified too soon or too often" [2, p. 216]. A proxy pattern might be useful in this case.

Notice that subject accepts a message to pass on to observers. The subject is therefore an intermediary between the dependents and a client who manages the data that undergoes change. Who is responsible for initiating the notification message? This is an open design decision. According to Cooper:

> If the Subject notifies all of the Observers when it is changed, each client is not responsible for remembering to initiate the notification. However, this can result in a number of small successive updates being triggered. If the clients tell the Subject when to notify the other clients, this cascading notification can be avoided; however, the clients are left with the responsibility to tell the Subject when to send the notifications. If one clients "forgets," the program simply won't work properly. [2, pp. 216–217]

We therefore have a tradeoff between potential cost of updates versus the ease of decoupling responsibilities. As in many situations, the best choice depends on the likely use pattern of the problem at hand.

4.4 Visitor

The visitor pattern decouples an algorithm from the object containing the data used by that algorithm. As Cooper [2] notes, this "turns the tables on our OO model and creates an external class to act on data in other classes" (p. 249). Essentially, the pattern adds methods to an existing class without actually modifying the class. Sometimes this happens because classes are final, operations weren't considered at design time, or the added methods are not directly meaningful or appropriate to the original class. This allows us to gather related operations into a single class, rather than modifying or extending existing classes to add operations.

Deploying the visitor pattern does require one minor change to the class containing the data. We must add an `accept` method to the visited class that takes a `Visitor` object:

```
public void accept(Visitor v)
{
  v.visit(this);
}
```

(If the original class is fixed, the method can be added via inheritance or composition.) The `accept` method simply passes control to a `Visitor` object. It turns out that is the crucial step in determining how to the visitor should handle the object being visited.

As you know, every Java object has both a static type, determined at compile time, and a dynamic type, which can only be determined at runtime. When the `accept` method is called for an object, Java uses dynamic dispatch to determine the runtime type of the visited object, and therefore the appropriate implementation of `accept`. When the visited object returns the call to `visit`, the static type of the visited object in that call is the runtime type of the original `accept` recipient, which may not have been visited variable's static type.

Let's consider the following simple example. We have four classes representing drawable shapes, a root `Shape` class, `Circle` and `Rectangle` which are derived from `Shape`, and `Square` which is naturally derived from `Rectangle`. The shapes themselves should not be responsible for handling their painting on the screen, so we give this responsibility to a `ShapePainter` class, which then is responsible for figuring out what to do with each shape.

```
public class ShapePainter implements Visitor
{
```

```
  public void visit(Circle c) { /* Draw a circle using c */ }
  public void visit(Rectangle r) { /* Draw a rectangle using r */ }
  public void visit(Square s) { /* Draw a square using s */ }
```

It is very important that *every class derived from* `Shape` *override the* `accept` *method exactly as shown above*. If you rely on the inheriting behavior, the appropriate static type cannot be passed to the visitor. This raises an important limitation of the visitor pattern. The `Visitor` interface must declare a `visit` method for every type of object in the hierarchy that is to be visited. This constraint often dovetails reasonably with the fact that a visitor is used when a set of classes is fixed and cannot be modified. Thus, we need not worry about expanding the visitor to handle additional classes.

Consider a polymorphic reference to a `Circle` object via a `Shape` reference, as in

```
Shape thing = new Circle();
```

If our `ShapePainter` is to draw `thing`, it must visit the shape with the call

```
thing.accept( new ShapePainter() );
```

While the static type of `thing` is a `Shape`, its runtime type is a `Circle`. Because the `Circle` class overrides the `accept` method as shown above, the reference to `this` in the `v.visit(this)` call must be (statically) a `Circle`, rather than a `Shape`.

We conclude our description of this pattern with a small example adapted from Palsberg and Jay [8], demonstrating a Scheme-style linked list of integers. Recall that in Scheme a list is either empty (called the null or nil list) or it is a pair (cons cell) whose car (head) is the value and cdr (tail) is a list.

```
public interface List
{
  public void accept(Visitor v);
}

public class Nil implements List
{
  public void accept (Visitor v) { v.visit(this); }
}

public class Cons implements List
{
  private int  head;
  private List tail;
  public  void accept (Visitor v) { v.visit(this); }
  public  int  getHead() { return head; }
  public  List getTail() { return tail; }
}
```

The classes `Nil` and `Cons` above represent the visited classes. Notice that both override `accept`. Next we must declare the `Visitor` interface for this family, which will include `visit` methods for both types of `List`s.

```
public interface Visitor
{
  public void visit (Nil  x);
  public void visit (Cons x);
}
```

We might do many things to a list of integers, such as take the sum or product, find the max, etc. We can encapsulate these operations in separate visitor classes. For instance,

```
public class SumVisitor implements Visitor
{
  private int sum = 0;

  public void visit (Nil x) { /* Nothing to do */ }

  public void visit(Cons x)
  {
    sum += x.getHead();
    x.getTail().accept(this);
  }

  private int getSum() { return sum; }
}
```

Notice that the `visit` method describes both the sum action and the access pattern of asking the cons cell's tail to accept the visitor. Assuming the variable `myList` refers to a `List` already defined elsewhere, we could calculate and print it sum as follows

```
SumVisitor sumListVisitor = new SumVisitor();
myList.accept(sumListVisitor);
System.out.println("The sum is " + sumListVisitor.getSum() );
```

In summary, while the visitor pattern does not give you access to private data, it can collect "data from a disparate collection of unrelated classes and utilize it to present the result of a global calculation to the user program" [2, p. 258].

Visitors can be troublesome when class hierarchies are still developing. As we noted earlier, the `Visitor` interface requires a method for each object of the hierarchy. When a new class is added, the corresponding method must be added to the `Visitor` interface *as well as* any concrete visitors that have already been implemented. However, once the class hierarchy has stabilized, the visitor pattern can be very helpful and useful [2, p. 258].

4.5 State

When you programmed in imperative languages, you likely represented discrete states of a program with an integer or an enumerated type. The problematic limitations of this representation of state arise when you need to add new states: every piece of code that checks the state variable will likely need to be changed. Fortunately, because of polymorphism, we can use an object to represent the program state. Switching state is as easy as changing the object (or changing its runtime type). The biggest benefit, however, arises from the fact that the implications of being in a particular state can be tightly coupled to the state object, rather than the more general program. That is, the behavior is part of the state, rather than whatever thing possesses that state.

Practically speaking, each state the program may be in must be represented by a concrete subclass of some root state type. This eliminates the need for long switch statements because the state-conditional behavior is handled by polymorphic references and dynamic dispatch. While this may end up creating a large number of very small classes, it greatly simplifies the program and reduces opportunities for error.

Bibliography

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995.

[2] James W. Cooper. *Java Design Patterns*: *A Tutorial*. Addison-Wesley, 2000.

[3] Builder pattern. Wikipedia, Retrieved 19 January 2010 from `http://en.wikipedia.org/wiki/Builder_pattern`

[4] Factory method pattern. Wikipedia, Retrieved 19 January 2010 from `http://en.wikipedia.org/wiki/Factory_method_pattern#Descriptive_names`

[5] William B. Sanders and Chandima Cumaranatunge. *ActionScript 3.0 Design Patterns Object Oriented Programming Techniques*. O'Reilly, 2007.

[6] Mark Allen Weiss. *Data Structures & Problem Solving Using Java (Fourth Edition)*. Addison Wesley, 2009.

[7] String (Java Platform SE 6). Retrieved 19 January 2011 from `http://java.sun.com/javase/6/docs/api/java/lang/String.html#intern()`

[8] Jens Palsberg, and C. Barry Jay. *The essence of the Visitor pattern*. In *Proceedings of The Twenty-Second Annual International Computer Software and Applications Conference*, pp. 9–15, 1998. doi: 10.1109/CMPSAC.1998.716629